Emulating TIE Instructions and TIE-Based Functions in MATLAB

Application Note

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
(408) 986-8000
Fax (408) 986-8919
www.tensilica.com

November 2009 Doc Number: AN09-240-00

© TENSILICA INC.

© 2009 Tensilica, Inc.

Printed in the United States of America

All Rights Reserved

This publication is provided "AS IS." Tensilica, Inc. (hereafter "Tensilica") does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Information in this document is provided solely to enable system and software developers to use Tensilica processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder to design or fabricate Tensilica integrated circuits or integrated circuits based on the information in this document. Tensilica does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Tensilica is a registered trademark of Tensilica, Inc. The following terms are trademarks of Tensilica, Inc. FLIX, OSKit, Sea of Processors, TurboXim, Vectra, Xenergy, Xplorer, and XPRES. All other trademarks and registered trademarks are the property of their respective companies.

Notice

Tensilica, Inc. reserves the right to make changes to its products or discontinue any of its products or offerings without notice.

Tensilica warrants the performance of its products to the specifications applicable at the time of sale in accordance with Tensilica's standard warranty.

Document Change History:

Published November 2009

Digitally signed by Tensilica
Technical Publications
Reason: Certified original
document from Tensilica

Contents

1 Introduction 1
Tool Versions 1
2 Possible Development Methodology 1
3 Compiling the Cstub Library 3
Working on 64-bit Machines 4
4 Modifying the MATLAB MEX Compilation and Linking Controls 4
Working on 64-bit machines 4
5 Wrapping TIE Instruction Emulation for MATLAB 4
Example 1: 24-bit Signed Fixed-Point Saturating Addition Instruction 5
MATLAB Code 5
C MEX-File Function for TIE AE_ADDSP24S 6
MATLAB Function to Wrap AE_ADDSP24S C MEX-File 9
MATLAB Results 13
Example 2: 24-bit Signed Fixed-Point Saturating Multiply Instruction 15
MATLAB Code 15
C MEX-File Function for TIE AE_MULFP24S_LL 16
MATLAB Function to Wrap AE_MULFP24S_LL C MEX-File 17
MATLAB Results 19
6 Wrapping TIE-Based Functions for use in MATLAB 20
Example: Simple Integer Vector Sum 20
MATLAB Code and Results 20
C MEX-File Function for Integer Vector Sum AE_FL_VEC_SUM_32 22
7 Summary 24
8 References 24

Code Examples

Code 1. AE_ADDSP24S.CPP File 6
Code 2. MAT_ADDSP24S Example 10
Code 3. MAT_ADDSP24S_Show Example 11
Code 4. AE_MULFP24S_LL,CPP Example 16
Code 5. MAT_MULFP24S_LL_SHOW Example 18
Code 6. AE_FL_VEC_SUM_32,CPP Example 22

© TENSILICA, INC.

© TENSILICA, INC.

Abstract

This application note describes how to emulate TIE functions and C-based function libraries written using TIE functions for a particular Xtensa target processor within the Mathworks MATLAB tool. The mechanism uses Cstubs (or Cstub functions), produced by the TIE compiler to allow host-based emulation of TIE functions. These Cstub functions are referenced by user-written C MEX-file functions that provide a wrapper for the interface between the Cstub files and MATLAB. The examples in this document are based on the HiFi 2 TIE package and Instruction Set Architecture (ISA).

© TENSILICA, INC.

1 Introduction

Emulating TIE functions with host-based execution is a good way to allow gradual, manual refinement of algorithms captured in MATLAB into towards optimized C code targeted to a particular Tensilica Xtensa configuration. Since many algorithms in MATLAB dataplane applications use fixed-point data types (or start as floating point, and then are converted to fixed point types), the conversions required and the interface code that needs to be written may be complicated. The interface between TIE host-based emulation (via the Cstub capability) and MATLAB is provided by C MEX-file functions that must be written for each TIE instruction that is desired, or for each function in a TIE-based function library. This application note provides several commented examples illustrating how to do so.

This document note uses the HiFi 2 TIE package as used in the Tensilica Standard 330HiFi Audio DSP because it is available as a Tensilica TIE package and illustrates the use of fixed-point data types.

Tool Versions

This application note was developed using version R2008b of MATLAB, and the RC-2009.0 version of Tensilica technology. The example TIE instructions and TIE function libraries are based on the Tensilica Standard 330HiFi DSP that includes the HiFi 2 TIE instruction extensions. This document uses the C++ version of the Cstub library, a Tensilica recommendation because of its maximum compatibility with various TIE constructs.

Note that this document was developed using Red Hat Linux; all commands shown are based on Linux.

The examples described herein make use of MATLAB fixed-point types, and thus require the availability of the MATLAB fixed-point toolbox. However, if your TIE instructions do not require the use of fixed-point types, this toolbox is not needed. Alternatively you may use your own specialized interpretations of standard C types such as shorts and ints to emulate fixed-point.

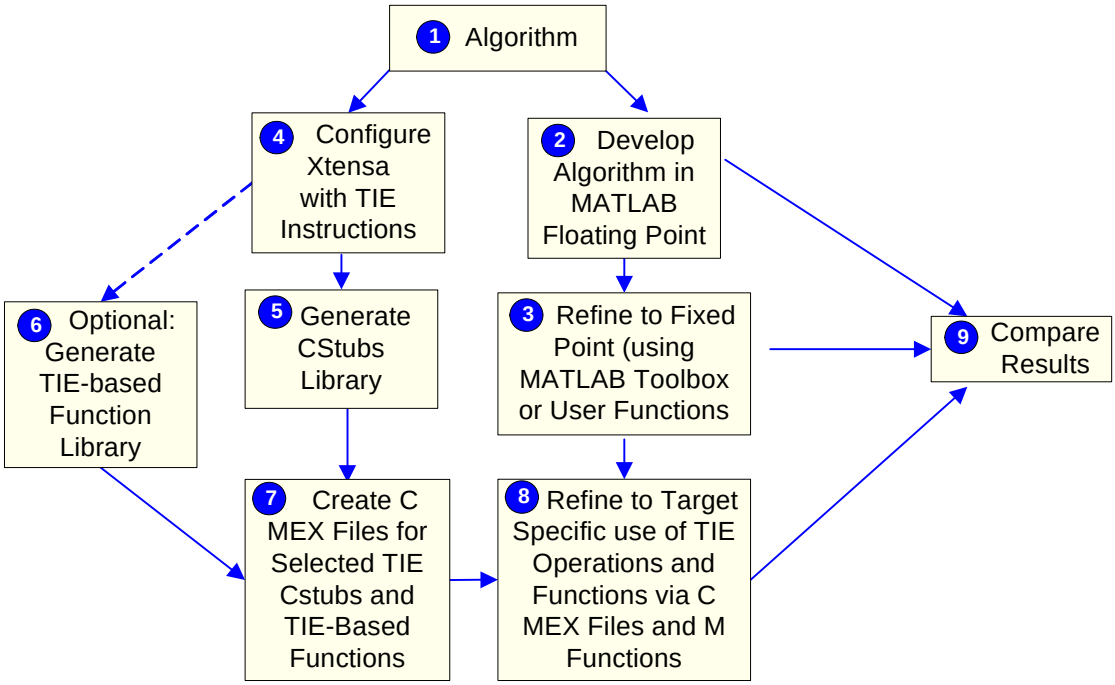
2 Possible Development Methodology

There are many development methodologies possible using the Mathworks tools MATLAB and Simulink. The focus of this application note is on emulating Tensilica TIE instructions for particular target processors from within MATLAB, using the Tensilica Cstub technology for host-based emulation of target TIE functions, and the MATLAB C MEX-file integration technique. This same method can also be used to build function libraries that make use of target TIE instructions, and call them from within MATLAB using the C MEX-file integration method. In addition, you can integrate complete applications or portions of applications written using Cstubs within MATLAB.

MEX is a capability offered by the Mathworks and stands for "MATLAB Executable". MEX-files are dynamically linked functions produced from C, C++ or Fortran that are compiled with the MEX compiler. They can then be run from within MATLAB just like any M-code or M-file (M is the native interpreted language of MATLAB). Various facilities in MEX allow data to be transferred to and from MATLAB to these MEX functions

© TENSILICA, INC.

The following figure illustrates a possible development methodology:


1 Algorithm
4 Configure Xtensa with TIE Instructions
2 Develop Algorithm in MATLAB Floating Point
6 Optional: Generate TIE-based Function Library
5 Generate CStubs Library
3 Refine to Fixed Point (using MATLAB Toolbox or User Functions
9 Compare Results
7 Create C MEX Files for Selected TIE Cstubs and TIE-Based Functions
8 Refine to Target Specific use of TIE Operations and Functions via C MEX Files and M Functions

The illustrated methodology is based on algorithm refinement in MATLAB, from floating point to fixed point to target-specific MATLAB code. This is done while simultaneously refining a Tensilica processor configuration to add instructions that will optimize the algorithm on a Tensilica processor. A variant of this methodology could be one where the processor configuration is fixed (e.g., a pre-configured Xtensa processor) and the algorithm developer is interested in porting to that particular Xtensa configuration.

Following are brief descriptions of the methodology steps shown above:

1. Start with an algorithm that you wish to model in MATLAB and implement on a Tensilica processor with instruction extensions to optimize the algorithm. This methodology example uses fixed-point instructions, which is only for illustration.
2. Implement the algorithm in MATLAB default double precision floating point.
3. Refine the algorithm in MATLAB into fixed point, using, for example, the MATLAB fixed-point toolbox. Here you would use toolbox features to define appropriate fixed-point types that match those being considered for use on the target Tensilica processor. The quantization, overflow, rounding, and saturation effects of using fixed-point types and arithmetic can be compared to floating point and the algorithm adjusted accordingly.
4. Simultaneously, either configure a Tensilica processor with appropriate TIE instructions, or use a pre-configured set of TIE instructions.
5. Part of the Xtensa deliverables is a Cstubs library matching your TIE instructions suitable for host-based emulation.
6. Optionally, develop more complex TIE-based functions as part of a function library, that within them uses TIE instructions. These function libraries may combine TIE instructions with other C/C++ code that implement useful reusable mathematical functions (e.g., an FFT routine). Such functions may hide some of the details of data type conversions, for easier code porting.
7. For the subset of TIE instructions and optional TIE-based functions, create C MEX-file wrappers that can interface with MATLAB and execute on the host workstation.

© TENSILICA, INC.

8. Further refine your MATLAB fixed-point algorithm and substitute the use of C MEX-file TIE and TIE-based functions for the native MATLAB fixed point functions. Use of C MEX-files will move your input data from MATLAB to the emulated TIE instructions and return the results in appropriate formats. Sometimes, if complex data conversions are required, it may be better to create MATLAB M-functions that incorporate the use of the C MEX-files and carry out data conversions.
9. You can then compare the results of each version of the algorithm: MATLAB floating point, fixed-point MATLAB, and target Tensilica emulated TIE operations and TIE-based functions. This ensures that the refined algorithm is functionally correct and produces results that within rounding, truncation, and quantization noise limits, is numerically correct.

From this M-code, which now mimics target Xtensa code invoking TIE functions, it should then be possible to manually refine it with the proper TIE function calls. The next step is to manually modify the code so that the algorithm can be compiled and executed on the Instruction Set Simulator (ISS) of the Tensilica processor, on a prototype of the processor, or a full hardware simulation or implementation. Note that these tasks are included in the *Linking MATLAB to Tensilica Instruction Set Simulator (ISS)* application note.

3 Compiling the Cstub Library

Information about the Cstub host-based simulation of TIE instructions are in the "Simulating TIE Instructions in a Native Environment" chapter of the *Tensilica Instruction Extension (TIE) Language User's Guide*.

For maximum compatibility with TIE constructs, use the C++ version of the Cstub library for host-based simulation of target TIE instructions (as recommended in the *Tensilica Instruction Extension (TIE Language User's Guide)*. It is possible that the C version of the Cstub library may work as an alternative, although there are use limitations that may or may not be relevant.

Use the "<Cstub.h>" header file and the "<Cstub.cpp>" source file. Also, use, the "<cstub-extif.h>" file that contains the definition of certain external interfaces (if generated for the particular configuration).

In general the .cpp and .h files are named cstub< XTENSA CONFIGURATION>.h, .cpp.

Copy these from the <XTENSA CONFIGURATION>/src/cstub directory to a working directory, and compile the Cstub library using the native host g++:

```
g++ -c <cstub.cpp>
```

For example, for the DC_B_330HiFi, the three files are:

- cstub-Xm_DC_B_330HiFi.h
- cstub-Xm_DC_B_330HiFi.cpp
- cstub-extif.h

The compile command is:

```
g++ -c cstub-Xm_DC_B_330HiFi.cpp
```

Create a library by converting the output .o file, using the "ar" Linux command, as follows:

```
ar –cr lib<Cstub library name>.a <Cstub library name>.o
```

For example, for the DC_B_330HiFi library, the command is:

```
ar –cr lib cstub-Xm_DC_B_330HiFi.a cstub-Xm_DC_B_330HiFi.o
```

This library file is then used in MATLAB with the MEX command to link C MEX-file headers that utilize the Cstub compiled TIE instruction emulation commands.

For these examples, a working directory, cCstubs_libs/hifi2, has been created to contain the libraries and include files for the TIE Cstubs.

© TENSILICA, INC.

Working on 64-bit Machines

Because the Cstub libraries are emulating a 32-bit target processor, the libraries only work in 32-bit mode on the host. Therefore, on a 64-bit machine, the libraries must be compiled with the 32-bit compilation option. For example, the –m32 option should compile Cstubs in 32-bit mode on a 64-bit machine with gcc or g++. Getting these to work with your version of MATLAB on a 64-bit machine is rather complicated; consult the Mathworks documentation or Mathworks for the best recommendation on how to do this.

4 Modifying the MATLAB MEX Compilation and Linking Controls

MEX compilation in MATLAB can be modified on the command line or by creating a user mexopts.sh file, as explained in Chapter 3 of the *MATLAB 7 External Interfaces* manual using the standalone `mex –setup` command option. Following is an illustration of MEX compilation controls with command line options.

Many of the MEX command-line options are very similar to those for gcc and g++. For example, to specify include directory paths and library files for linking, the same `–I` and `–L` options are used.

To compile an example C MEX file for a TIE instruction, use the following command line in MATLAB:

```
mex  <C Mex-file function name.cpp> -I<directory location of Cstub header file>
-D__XTENSA_HIFI2__ -L<directory location of Cstub library> -lcstub-
XmDC_B_330HiFi
```

Working on 64-bit machines

Because the Cstub libraries only work in 32-bit mode on the host, they cannot be used with a 64-bit version of MATLAB on a 64-bit machine. However, Mathworks in its documentation and online support database does discuss how to link 32-bit libraries with MATLAB and run it on a 64 bit machine. Please consult Mathworks and its documentation for up-to-date guidance.

5 Wrapping TIE Instruction Emulation for MATLAB

The examples in this document are drawn from the HiFi 2 TIE instructions used in the 330HiFi example target configuration. These examples use a variety of fixed-point types, and thus use the MATLAB fixed-point toolbox to illustrate and compare the use of MATLAB built-in double precision floating point, MATLAB fixed point, and TIE instruction emulation using C MEX-file functions.

In general, each example shows several ways of performing calculations, including:

- MATLAB double precision floating point
- Fixed point MATLAB toolbox
- TIE emulation with the same fixed point variables as used in MATLAB

To transfer these variables to and from the TIE emulation C MEX-file functions, they will be cast or converted into appropriate signed integer types, which will be used as “carriers” for the bit patterns representing the fixed-point values.

These examples may need only a small amount of modification to be adapted to your use of TIE, other TIE packages, or pre-configured DSP options and libraries. However, depending on various TIE constructs you use, the examples may require extensive modification. The HiFi 2 ISA is used here for illustrative purposes only.

© TENSILICA, INC.

Example 1: 24-bit Signed Fixed-Point Saturating Addition Instruction

Example 1 illustrates the use of the HiFi2 AE_ADDSP24S instruction. This instruction adds two 24-bit signed fixed-point numbers together. The representation used is 1.23 (also called 1Q23, Q23 or Q1.23)—that is, one sign bit and 23 bits for the fractional part of the 24-bit word. Thus the values represented are in a range from [-1.0 to +1.0). The instruction is saturating; adding +0.6 to +0.5 in saturating arithmetic gives a result of +1.0. (Note that quantization in fixed point may mean that the representation of 0.6, 0.5 and 1.0 may differ slightly).

Please note that the HiFi2 AE_ADDSP24S instruction actually does two additions, of two pairs of 24-bit signed fixed-point numbers that have been stored in 48-bit TIE AE_PR registers. Many of the HiFi 2 TIE instructions carry out operations on pairs of variables. The conversion functions used to transfer 24-bit variables from MATLAB to the C MEX-files will end up duplicating the two 24-bit numbers in the equivalent in Cstubs of a 48-bit register, and the addition will add both of these; but the wrapper code in this example is set up to return only one result. Of course, this example could be modified to send four 24-bit numbers across, load them appropriately, and return two results; but we choose here to implement the simpler form to avoid additional complexity. See the *HiFi 2 Audio Engine Instruction Set Architecture Reference Manual* for more details about AE_ADDSP24S.

Note that in order to see the effects of quantization, the numeric format in MATLAB was set to “long” (so that 16 digits after the decimal point are displayed, rather than the default of 4).

This example uses the MATLAB fixed-point toolbox.

MATLAB Code

First, we declare the basic fixed point numeric type for 24-bit signed fixed point.

This is a MATLAB fixed point type, signed, with a word length of 24-bits and a fractional part of 23 bits (the first bit is for the sign).

```
s24_23=numerictype(1,24,23)
```

We declare two MATLAB variables, p0 and p1, which by default will be double precision floating point, and add them together:

```
p0 = -.7789325
p1 = .5016242
p = p0+p1
```

We declare two fixed point variables of the s24_23 type and set their value to p0 and p1. We also add them together and convert the type to s24_23 (otherwise, MATLAB will declare a larger fixed point type to handle a larger result). This uses the MATLAB fixed point toolbox “fi” constructor.

```
q0 = fi(p0,s24_23)
q1 = fi(p1,s24_23)
q = fi(q0+q1,s24_23)
```

We call a MATLAB function, “mat_addsp24s”, which wraps our C MEX-file function that implements a call to the Cstub AE_ADDSP24S TIE instruction:

```
r = mat_addsp24s_show(q0, q1)
```


© TENSILICA, INC.

Here we compare the original floating point result to the MATLAB fixed point result and also the MATLAB fixed point result to the target TIE code result:

```
err_float_fix = double(abs(p-q))
err_float_target = double(abs(p-r))
err_fix_target=double(abs(q-r))
```

The following uses a saturating add:

```
a = fi(.6,s24_23)
b = fi(.5,s24_23)
q = mat_addsp24s_show(q0, q1)
```

C MEX-File Function for TIE AE_ADDSP24S

The C MEX-file function code for the TIE function AE_ADDSP24S acts as a wrapper for the Cstub AE_ADDSP24S implementation. This function code carries out basic error checking of the input and output arguments passed from MATLAB to C, and the appropriate extraction of operands, conversion operands and insertion of the result into the structures used to pass information between MATLAB and C.

The code, with embedded comments, follows. This code, compiled with the MEX command, is in the AE_ADDSP24S.cpp file, contained in the MATLAB working directory:

```
mex AE_ADDSP24S.cpp –I<DIRECTORY>/Cstubs_libs/hifi2 -D__XTENSA_HIFI2__ -
L<DIRECTORY>/Cstubs_libs/hifi2 –lcstub_Xm_DC_B_330HiFi
```

Following is the contents of AE_ADDSP24S.cpp (with inserted comments):

CODE 1. AE_ADDSP24S.CPP FILE

```
#include<stdio.h>
//  the Cstub header file
#include <cstub_Xm_DC_B_330HiFi.h>

//  Including “mex.h” is required for all C Mex-file functions
#include "mex.h"

/* Input Arguments */
//   MATLAB uses pointers to structures for “rhs” = right-hand-side
//       arguments for inputs, and “lhs” = left-hand-side arguments
//       for outputs

#define   P0  prhs[0]
#define   P1  prhs[1]

/* Output Arguments */

#define   P   plhs[0]

//   MATLAB uses matrices for inputs and outputs.  In this example,
```


© TENSILICA, INC.

```
//         the output matrix has 1 dimension.

#define NDIMS 1

//   This is a wrapper function (a "computational subroutine")
//      for the TIE Cstub AE_ADDSP24S
//      function.   We use a different name for the wrapper to
//      avoid confusion

int addsp24s(const int * inp0, const int *  inp1)
{
//   the input arguments are passed in as pointers so we
//        de-reference them into 32 bit integers.
//        We use a 32 bit temporary variable "outval"
//        to return the result.
    int i0, i1, outval;
    i0 = *inp0;
    i1 = *inp1;

   /* Call the TIE Cstub AE_ADDSP24S function */
    ae_p24x2s input0 = *((ae_p24s *)&i0);
    ae_p24x2s input1 = *((ae_p24s *)&i1);

    ae_p24x2s output = AE_ADDSP24S(input0, input1);


    *((ae_p24s *) &outval) = output;
    return outval;
}

//     The standard C Mex-file function is always called
//         mexFunction.
//         Note that two mxArrays are declared, one for right hand
//         side input arguments, and one for left hand side
//         output arguments.  Further, the nrhs and nlhs declare
//         the number of input and output arguments used in the
//         MATLAB call to the function.
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    int i;
    int *out;
    int *inp0,*inp1;
```


© TENSILICA, INC.

```
//  The output argument array is a single variable, so
//       the dims array is defined as dims[0] = 1.
    mwSize dims[]={1};

// Check number right hand side arguments is 2.
    if (nrhs != 2)
      mexErrMsgTxt("Must have 2 right-hand-side arguments.\n");

//  Check number of left hand side arguments is 1.
    if (nlhs != 1)
      mexErrMsgTxt("Must have 1 left-hand-side argument.\n");


   /*  Check that both input arguments are INT32 type  */
    if ((!mxIsClass(P0, "int32")) || (!mxIsClass(P1, "int32")))        {
   mexErrMsgTxt("AE_ADDSP24S requires that both input arguments be 32 bit
   integers");
        }

//   We declare a “numeric array”, of type INT32, to hold the
//       output.  Note that it is a 1x1 array.
    P = mxCreateNumericArray(NDIMS, dims, mxINT32_CLASS, mxREAL);

/* Assign pointers to the various parameters */
//   The mex function mxGetPr gets a pointer to the input and
//        output arrays as needed.


    out = (int*) mxGetPr(P);


    inp0 = (int*) mxGetPr(P0);
    inp1 = (int*) mxGetPr(P1);

/*   call a subroutine to actually do the computation  */

//    This fills the output array with the value determined by the
//        call to the computational subroutine which actually
//        calls the TIE Cstub function.

   (*out) = addsp24s(inp0, inp1);
}
```


© TENSILICA, INC.

MATLAB Function to Wrap AE_ADDSP24S C MEX-File

The C MEX-file for AE_ADDSP24S can be called directly from our MATLAB algorithmic M-code. However, there are some rather complex type conversions involved that if implemented directly in the top-level version of the algorithm, would make it complex to read and might obscure what the algorithm is doing. MATLAB allows you to create MATLAB functions that can hide some of the data conversion details, and in addition, do some error checking before calling a C MEX-file. Of course, MATLAB M-code functions can be used in general in developing in MATLAB. This document illustrates the use of an intermediate MATLAB function, mat_addsp24s, to wrap the direct call to the C MEX-file AE_ADDSP24S.cpp.

In MATLAB, if the interpreter is told to invoke a function, it looks first for a MATLAB M-function with the name and then for a compiled C MEX-file with the same name appended with ".mexglx". Since we do not have a MATLAB M-code function called AE_ADDSP24S, but do have a compiled .mexglx function with that name, MATLAB will find the correct function when it runs.

Following is the MATLAB wrapper function, mat_addsp24s. Note the semicolon (;) after each MATLAB statement. This suppresses the output that would normally come after each MATLAB statement is executed.

© TENSILICA, INC.

CODE 2. MAT_ADDSP24S EXAMPLE

```
function p = mat_addsp24s(p0, p1)
% MAT_ADDSP24S   Wrapper function for TIE cstub AE_ADDSP24S instruction
%   This function does a series of type conversions of operands  - input
%   args and output args.

%  Basic checking that number of input and output arguments is correct.
if (nargin ~= 2)
   error('Must have 2 input arguments.');
end
if (nargout ~= 1)
   error('Must have 1 output argument.');
end

%   Declare three fixed point types required.   s24_23 is used
%   for the output argument; the other two are used as intermediate
%   conversions.

s32_0=numerictype(1,32,0);
s32_23=numerictype(1,32,23);
s24_23=numerictype(1,24,23);

%   First convert the two inputs to int32

i32p0 = int32(p0);
i32p1 = int32(p1);

%   Call the TIE cstub AE_ADDSP24S instruction

ae_result = AE_ADDSP24S(i32p0, i32p1);

%   Convert the result to a s32_0

s32_0_result = fi(ae_result, s32_0);

%   Then do a fixed point reinterpretcast because this is actually
%   a (9.23) fixed point type

s32_23_result = reinterpretcast(s32_0_result,s32_23);

%  Finally, do a fixed point conversion to the final output and return

p = fi(s32_23_result,s24_23);
end
```


© TENSILICA, INC.

The following wrapper function, "mat_addsp24s_show", is a version of the above, but this one does not use the semicolons at the end of each line. Thus, Code 3 illustrates the results of executing each line of this function, which is printed out when we run the "_show" version of this function.

CODE 3. MAT_ADDSP24S_SHOW EXAMPLE

```
>> q = mat_addsp24s_show(q0, q1)
s32_0 =
          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 0
s32_23 =
          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 23
s24_23 =
          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 24
        FractionLength: 23
```

The above declares two signed fixed point types, both 32 bits long; the first has 0 fraction bits (in other words, it acts as a fixed point version of a signed 32-bit integer); the second has 23 fractional bits, so it is similar to the (9.23) type discussed in the *HiFi 2 Audio Engine Instruction Set Architecture Reference Manual*. Both these types are used in conversions from MATLAB to C MEX-file functions and back, as illustrated in the function code above. Note that the s24_23 fixed point type must be declared again in this function so that it can be used, due to MATLAB scoping rules.

```
i32p0 =
    -6534159
i32p1 =
     4207929
```

`i32p0` and `i32p1` are conversions of the two input variables, p0 and p1, into int32 types. The int32 type is used as a container to hold the bit pattern of the (1.23) types to pass from MATLAB to the C MEX-file function.

© TENSILICA, INC.

```
ae_result =
    -2326230
```

`ae_result` is returned from the C MEX-file function AE_ADDSP324S via the call

```
ae_result = AE_ADDSP24S(i32p0, i32p1);
```

Note that it contains the bit pattern of the result as an int32. Remember that the C MEX-file function shown declared the output variable to be of class `mxINT32_CLASS`.

```
s32_0_result =
    -2326230
```

The result of this addition is a signed 32-bit integer. It is brought back into MATLAB and converted to the signed fixed point type s_32_0.

```
s32_23_result =
  -0.277308225631714
```

The correct result of the AE_ADDP24 is actually a signed fraction, 23 bits long, contained in the least significant 23 bits of the 32-bit fixed point type. To extract this value correctly, we do a MATLAB reinterpret cast to convert the s_32_0 to an s_32_23 fixed point type. Note that when data types are not a power of 2, both MATLAB and TIE have specific rules about data formatting and the use of extra bits in the formats. To write a correct wrapper, it is necessary to have an understanding of the rules that apply in both MATLAB and TIE. In this example, the rules are the same — but note that is not always the case.

```
p =
   -0.277308225631714
          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 24
        FractionLength: 23

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

The final conversion will convert a fixed point, 32-bit signed with 23 bits fraction into a 24-bit signed fixed point type with 23 bits fraction. This is done with the line of code:

```
p = fi(s32_23_result,s24_23);
```

And the result, p, is returned as the output argument from this MATLAB function.

© TENSILICA, INC.

MATLAB Results

```
>>p0= -.7789325
p0 =
  -0.778932500000000
>> p1 = .5016242
p1 =
   0.501624200000000
>> p0+p1
ans =
  -0.277308300000000
```

The following code shows that the MATLAB double precision floating point has enough precision to handle the two values being entered, and produces a result that is accurate to the precision of double precision floating point.

```
>> q0=fi(p0,s24_23)
q0 =
  -0.778932452201843

          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 24
        FractionLength: 23

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

>> q1=fi(p1,s24_23)
q1 =
   0.501624226570129
>> q=fi(q0+q1,s24_23)
q =
  -0.277308225631714
```

Note that MATLAB normally prints out copious information about each fixed-point variable; it is illustrated for q0, but suppressed for the rest of the code. Note also that q0 and q1 do not exactly match the values of p0 and p1. This is due to the quantization effects of the s_24_23 fixed point data type being used. Thus the sum q also differs from the sum p.

© TENSILICA, INC.

In addition, it may be necessary to modify attributes of the data type declaration, such as its RoundMode, OverflowMode, etc., depending on the characteristics of the TIE functions that you are emulating and whether they are saturating or overflow, for example:

```
>> r = mat_addsp24s(q0, q1)
r =
  -0.277308225631714
```

The MATLAB function, "mat_addsp24s", which wraps our C MEX-file function that implements a call to the Cstub AE_ADDSP24S TIE instruction, returns a result that is identical to that using the MATLAB fixed-point toolbox:

```
>> err_float_fix = double(abs(p-q))
err_float_fix =
     5.960464477539062e-08
>> err_float_target = double(abs(p-r))
err_float_target =
     5.960464477539062e-08
>> err_fix_target=double(abs(q-r))
err_fix_target =
     0
```

The above code shows that the quantization involved in converting the p0 and p1 inputs to q0 and q1, and the fixed point add involved in computing q, means that there is a difference between the floating point and fixed point results, which we illustrate as err_float_fix. We also compute the error between the floating point and the target (TIE emulated computation) as err_float_target. Note in particular that err_float_fix = err_float_target, thus there is no difference between the MATLAB fixed point result and the emulated TIE result, which is of course what we would expect.

```
>> a = fi(.6,s24_23)
a =
   0.600000023841858
>> b = fi(.5,s24_23)
b =
   0.500000000000000
>> q = mat_addsp24s (q0, q1)
q =
   0.999999880790710
```

Above we only try the MATLAB function that wraps the TIE instruction. Note that quantization means that a is not quite the input +0.6, but b is exactly +0.5. The result is as close to +1.0 as the (1.23) data type quantization allows.

© TENSILICA, INC.

Example 2: 24-bit Signed Fixed-Point Saturating Multiply Instruction

Example 2 illustrates the use of the HiFi 2 AE_MULFP24S_LL instruction. This instruction multiplies two 24-bit signed fixed-point numbers together. The representation used for the inputs is (1.23)—that is, one sign bit and 23 bits for the fractional part of the 24-bit word. Thus the values represented are in a range from -1.0 to +1.0. The instruction is defined in the *HiFi 2 Audio Engine Instruction Set Architecture Reference Manual.* The output is a signed 56-bit fixed point (9.47) representation. Since the two inputs range from -1 to +1, the output of the multiply will also range between -1 and +1.

Please note that the HiFi2 AE_MULFP24S_LL instruction actually does one multiplication of the L elements contained in two 48-bit TIE AE_PR registers, which contain two 24-bit signed fixed point (1.23) numbers. (There are other variant HiFi 2 instructions for multiplying HH, HL and LH elements). The conversion functions used to transfer 24-bit variables from MATLAB to the C MEX-files will end up duplicating the two 24-bit numbers in the equivalent in Cstubs of a 48-bit register, and the multiplication will pick out the L elements of each. See the *HiFi 2 Audio Engine Instruction Set Architecture Reference Manual* for more details about AE_ MULFP24S_LL and its variants..

Note that in order to see the effects of quantization, the numeric format in MATLAB was set to “long” (so that 16 digits after the decimal point are displayed, rather than the default of 4).

This example uses the MATLAB fixed-point toolbox.

The structure of this example is similar to Example 1, so fewer comments are used.

MATLAB Code

First, we declare the basic fixed point numeric type for 24-bit signed fixed point, and a fixed point type for 56 bit signed (9.47) results.

```
s24_23=numerictype(1,24,23)
s56_47=numerictype(1,56,47)
p0 = -.7789325
p1 = .5016242
p = p0*p1
```

We declare two MATLAB double precision floating point variables, p0 and p1, and multiply them.

We create fixed point versions of p0 and p1, and multiply them, using the 56 bit signed (9.47) fixed point type for the result:

```
q0 = fi(p0,s24_23)
q1 = fi(p1,s24_23)
q = fi(q0*q1,s56_47)
```

Following is the MATLAB function, “mat_mulfp24s_ll_show”, which wraps our C MEX-file function that implements a call to the Cstub AE_MULFP24S_LL TIE instruction:

```
r = mat_mulfp24s_ll_show(q0, q1)
```


© TENSILICA, INC.

Following, we compare the original floating point result to the MATLAB fixed point result; and also the MATLAB fixed point result to the target TIE code result.

```
err_float_fix = double(abs(p-q))
err_float_target = double(abs(p-r))
err_fix_target=double(abs(q-r))
```

C MEX-File Function for TIE AE_MULFP24S_LL

The C MEX-file function code for the TIE function AE_MULFP24S_LL acts as a wrapper for the Cstub AE_MULFP24S_LL implementation. This code carries out basic error checking of the input and output arguments passed from MATLAB to C, and the appropriate extraction of operands, conversion operands and insertion of the result into the structures used to pass information between MATLAB and C.

The code, with a few embedded comments, is compiled with the MEX command. It is contained in a file AE_MULFP24S_LL.cpp contained in the MATLAB working directory.

```
mex AE_MULFP24S_LL.cpp –I<DIRECTORY>/Cstubs_libs/hifi2 -D__XTENSA_HIFI2__ -
L<DIRECTORY>/Cstubs_libs/hifi2 –lcstub_Xm_DC_B_330HiFi
```

Following is the contents of AE_MULFP24S_LL.cpp (with inserted comments):

CODE 4. AE_MULFP24S_LL,CPP EXAMPLE

```
#include <cstub-Xm_DC_B_330HiFi.h>
#include "mex.h"

/* Input Arguments */
#define   P0  prhs[0]
#define   P1  prhs[1]

/* Output Arguments */

#define   P   plhs[0]

#define NDIMS 1
//
//   We use signed 64 bit integer variables to transmit the intermediate
//    results  (i.e., long long ints)
//
long long int mulfp24sll(int * inp0, int *  inp1)
{
    int i0, i1;
    long long int outval;
    i0 = *inp0;
    i1 = *inp1;
```


© TENSILICA, INC.

```
   /* Call the TIE cstub AE_MULFP24S_LL function */
    ae_p24x2s input0 = *((ae_p24s *)&i0);
    ae_p24x2s input1 = *((ae_p24s *)&i1);

    ae_q56s output = AE_MULFP24S_LL(input0, input1);

    *((ae_q56s *) &outval) = output;

    return outval;
}

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
    int i;
//  Since the output is contained in a 64 bit integer (int64),
//  we use a long long int pointer to it
    long long int *out;
    int *inp0,*inp1;
    mwSize dims[]={1};

    if (nrhs != 2) mexErrMsgTxt("Must have 2 right-hand-side
                                arguments.\n");
    if (nlhs != 1) mexErrMsgTxt("Must have 1 left-hand-side argument.\n");
    if ((!mxIsClass(P0, "int32")) || (!mxIsClass(P1, "int32")))
        mexErrMsgTxt("AE_MULFP24S_LL requires that both input arguments be
                    32 bit integers");

    P = mxCreateNumericArray(NDIMS, dims, mxINT64_CLASS, mxREAL);

    out = (long long int *) mxGetPr(P);

    inp0 = (int*) mxGetPr(P0);
    inp1 = (int*) mxGetPr(P1);

    out[0]=  mulfp24sll(inp0, inp1);
}
```

MATLAB Function to Wrap AE_MULFP24S_LL C MEX-File

Code 5 shows the MATLAB function, “mat_mulfp24s_ll_show”, which is used to wrap the call to the TIE AE_MULFP24S_LL function that is implemented in the AE_MULFP24S_LL.cpp C MEX-file. Note that this is a “show” version of the function, which will display results after each step, because there are no semicolons after each line. We also have the normal version of this function with semicolons after each line, “mat_multfp24s_ll”.

© TENSILICA, INC.

CODE 5. MAT_MULFP24S_LL_SHOW EXAMPLE

```
function p = mat_mulfp24s_ll(p0, p1)
% MAT_MULFP24S_LL   Wrapper function for TIE cstub AE_MULFP24S_LL instruction
%   This function does a series of type conversions of operands  - input
%   args and output args.

%  Basic checking that number of input and output arguments is correct.
if (nargin ~= 2)
   error('Must have 2 input arguments.')
end
if (nargout ~= 1)
   error('Must have 1 output argument.')
end

%   Declare three fixed point types required.   s56_47 is used
%   for the output argument; the other two are used as intermediate
%   conversions.

s64_0=numerictype(1,64,0)
s64_47=numerictype(1,64,47)
s56_47=numerictype(1,56,47)

%   First convert the two inputs to int32

i32p0 = int32(p0)
i32p1 = int32(p1)

%   Call the TIE cstub AE_MULFP24S_LL instruction

ae_result = AE_MULFP24S_LL(i32p0, i32p1)

%   Convert the result to a s64_0

s64_0_result = fi(ae_result, s64_0)

%   Then do a fixed point reinterpretcast because this is actually
%   a (17.47) fixed point type

s64_47_result = reinterpretcast(s64_0_result,s64_47)

%  Finally, do a fixed point conversion to the final output and return

p = fi(s64_47_result,s56_47)

end
```


© TENSILICA, INC.

The results of executing this function are shown earlier, and the detailed transformations are very similar conceptually to those in Example 1, however, Example 2 uses 64-bit integer data types and 64-bit fixed point data types, as well as 56 bit (9.47) fixed point types to deal with the output.

MATLAB Results

```
>>p0= -.7789325
p0 =
  -0.778932500000000
>> p1 = .5016242
p1 =
   0.501624200000000
>> p = p0*p1
p =
  -0.390731392166500
>> q0=fi(p0,s24_23)
q0 =
  -0.778932452201843

          …… (datatype information)

>> q1=fi(p1,s24_23)
q1 =
   0.501624226570129
>> q=fi(q0*q1,s56_47)
q =
  -0.390731388886124
>> r = mat_mulfp24s_ll (q0, q1)
r =
  -0.390731388886124
```

The MATLAB function, “mat_mulfp24s_ll”, which wraps our C MEX-file function that implements a call to the Cstub AE_MULFP24S_LL TIE instruction, returns a result that is identical to that using the MATLAB fixed-point toolbox.

```
>> err_float_fix = double(abs(p-q))
err_float_fix =
     3.280376081882252e-09
>> err_float_target = double(abs(p-r))
err_float_target =
     3.280376081882252e-09
>> err_fix_target=double(abs(q-r))
err_fix_target =
     0
```


© TENSILICA, INC.

6 Wrapping TIE-Based Functions for use in MATLAB

Example: Simple Integer Vector Sum

There is an alternative to directly calling TIE functions from your M-code, or refining M-code based algorithms in MATLAB into code that is target-optimized using the TIE Cstubs methods discussed earlier. Instead, you can create function libraries that internally incorporate TIE functions via Cstubs, but do not expose the TIE functions individually as in the earlier examples. These may use normal C or C++ types, or use such types as containers — for example, fixed-point types. You can use the C MEX-file methods discussed earlier to allow the invocation of such functions from MATLAB M-code. Depending on the complexity of the transformations, you can use an intermediate MATLAB M-code function as in the earlier examples to hide some of the type conversions. On the other hand, if you are using ordinary C types, it may be simpler to directly call the C MEX-file function.

This first C MEX-file function example, AE_FL_VEC_SUM_32, shows the use of a very simple function that uses TIE intrinsic instructions from HiFi 2 to sum two vectors of normal 32-bit integers, producing an output vector of the same size. Because there are no MATLAB fixed point types used here, the use of an intermediate MATLAB function is not necessary. Instead, this is illustrated by declaring some MATLAB one-dimensional arrays of integers and adding them together in MATLAB, and then sum them using the TIE-based function encapsulated in the C MEX-file.

Note that although the C MEX file contains the implementation of the TIE based function as a separate routine within a single source, it could instead reference a precompiled function library via a function prototype declared as external, and the linkage directives for MEX compilation could add that library to the list of libraries used.

MATLAB Code and Results

The following simple function shows only the MATLAB code and its results, followed by a commented C MEX-file listing.

First we declare two vectors of 10 integers each, add them together in MATLAB and using the AE_FL_VEC_SUM_32 C MEX-file, add them together using the emulation of the TIE-based function:

```
>> a = int32([1 2 3 4 5 -1 -2 -3 -4 -5])
a =
  Columns 1 through 5
           1           2           3           4           5
  Columns 6 through 10
          -1          -2          -3          -4          -5

>> b = int32([101:110])
b =
  Columns 1 through 5
         101         102         103         104         105
  Columns 6 through 10
         106         107         108         109         110

>> a + b
ans =
  Columns 1 through 5
         102         104         106         108         110
  Columns 6 through 10
```


© TENSILICA, INC.

```
         105         105         105         105         105

>> c = AE_FL_VEC_SUM_32(a,b)
c =
  Columns 1 through 5
         102         104         106         108         110
  Columns 6 through 10
         105         105         105         105         105
```

Note that the TIE-based C MEX-file function and MATLAB produce identical results. Next is an illustration of error handling in the C MEX-file: We first pass just one input argument in; then we try passing in two unequally-sized arrays; then finally, pass in two arrays of unequal type (one int32, the other default double precision floating point MATLAB variables).

```
>> c = AE_FL_VEC_SUM_32(a)
??? Error using ==> AE_FL_VEC_SUM_32
Two input (right-hand-side) arguments required.

>> b = int32([101:109])
b =
  Columns 1 through 5
         101         102         103         104         105
  Columns 6 through 9
         106         107         108         109

>> c = AE_FL_VEC_SUM_32(a,b)
AE_FL_VEC_SUM_32 requires that both input vectors have the same 1 X dimension .
The first is 1 X 10; the second 1 X 9
??? Error using ==> AE_FL_VEC_SUM_32
Will not evaluate

>> a = [1 2 3 4 5 -1 -2 -3 -4 -5]
a =
     1     2     3     4     5    -1    -2    -3    -4    -5

>> b = int32([101:110])
b =
  Columns 1 through 5
         101         102         103         104         105
  Columns 6 through 10
         106         107         108         109         110

>> c = AE_FL_VEC_SUM_32(a,b)
??? Error using ==> AE_FL_VEC_SUM_32
AE_FL_VEC_SUM_32 requires that both input vectors be 32 bit integers
```


© TENSILICA, INC.

C MEX-File Function for Integer Vector Sum AE_FL_VEC_SUM_32

Code 6 shows the contents of AE_FL_VEC_SUM_32.cpp with inserted comments:

CODE 6. AE_FL_VEC_SUM_32,CPP EXAMPLE

```
#include <cstub-Xm_DC_B_330HiFi.h>

#include "mex.h"

/* Input Arguments */

#define   INP1_IN   prhs[0]
#define   INP2_IN   prhs[1]

/* Output Arguments */

#define   VEC_SUM_OUT    plhs[0]
#define NDIMS 2

void vec_sum_32_hifi2(const int * __restrict__ inp1,const int * __restrict__
inp2, int * __restrict__ out, const int size)
{
   int i;
   ae_q56s input1,input2,output;

   for(i=0;i<size;i++){

      input1 = *((ae_q32s *)&inp1[i]);
      input2 = *((ae_q32s *)&inp2[i]);

      output = AE_ADDSQ56S(input1, input2);

      /* saturate Q to 1.47    */
      output = AE_SATQ48S (output);

      *((ae_q32s *) &out[i])=output;
   }

}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    int        i;
    mwSize m1, n1, m2, n2;
    unsigned int size;
    mwSize dims[]={1,0};
```


© TENSILICA, INC.

```
    int *vec_sum;
    int *inp1,*inp2;

    /* Check for proper number of arguments */
    if (nrhs != 2) mexErrMsgTxt("Two input (right-hand-side) arguments
required.");
    if (nlhs > 1) mexErrMsgTxt("One output (left-hand-side) argument
required");

    /* Check the dimensions of the 2 input vectors.  They must be the same size
*/
    /*   Note:   A row vector in MATLAB is a 1 X size vector; M = 1, N = size
*/

    m1 = mxGetM(INP1_IN);
    n1 = mxGetN(INP1_IN);

    m2 = mxGetM(INP2_IN);
    n2 = mxGetN(INP2_IN);

    if ((m1 > 1) || (m2 > 1) || (m1 != m2) || (n1 < 1) || (n2 < 1) || (n1 !=
n2))         {
       mexPrintf("AE_FL_VEC_SUM_32 requires that both input vectors have the
same 1 X dimension .\nThe first is %d X %d; the second %d X %d\n", m1, n1, m2,
n2);
        mexErrMsgTxt("Will not evaluate\n");
    }

   /*  Check that both input vectors are INT32 type  */
    if ((!mxIsClass(INP1_IN, "int32")) || (!mxIsClass(INP2_IN, "int32")))
    mexErrMsgTxt("AE_FL_VEC_SUM_32 requires that both input vectors be 32 bit
    integers");

    size = (unsigned int) n1;

    dims[1]=n1;

   /* Create a matrix for the return argument */

    VEC_SUM_OUT = mxCreateNumericArray(NDIMS, dims, mxUINT32_CLASS, mxREAL);

/* Assign pointers to the various parameters */
    vec_sum = (int*) mxGetPr(VEC_SUM_OUT);

    inp1 = (int*) mxGetPr(INP1_IN);
    inp2 = (int*) mxGetPr(INP2_IN);

/* Do the actual computations in a subroutine */
    vec_sum_32_hifi2(inp1, inp2, vec_sum, size );
}
```


© TENSILICA, INC.

7 Summary

In this application note we have discussed how to wrap host-based emulations of single TIE functions implemented using Cstubs so that they can be called from MATLAB models. This methodology can be extended so that complex functions using many calls to Tensilica instructions emulated with Cstubs can also be wrapped and called. Finally, the methodology can be extended to allow users to call complete emulated applications or major portions of applications, targeted to Tensilica processor configurations, to be called from MATLAB. Given the flexibility of MATLAB and its wide variety of methods and interfaces, many alternatives to this methodology are possible.

8 References

Tensilica recommends the following documents for additional information:

- *Tensilica Instruction Extension (TIE) Language User's Guide*: The "Simulating TIE Instructions in a Native Environment" chapter
- *MATLAB 7 External Interfaces Manual*. Chapter 3 : « Calling C and Fortran Programs from MATLAB command line » ; Chapter 4 : « Creating C Language MEX-files"
- *MATLAB Fixed Point Toolbox 2 User's Guide*
- *HiFi2 Audio Engine Instruction Set Architecture Reference Manual*.

© TENSILICA, INC.